SOLID Principles for Object-Oriented Design

- CSCE 315: Programming Studio
- Instructor: Yoonsuck Choe
- See the URL above for diagram notations.
- Topic motivated by Chris Weldon @ Improving.

History of SOLID

Robert C. Martin is the main person behind these ideas (some individual ideas predate him though).
- First appeared as a news group posting in 1995.
- Lots of online learning material (find on your own).

Benefits of SOLID

- Provides a principled way to manage dependency.
- Serves as a solid foundation for OOD upon which more complicated design patterns can be built upon and incorporated naturally.
- Results in code that are flexible, robust, and reusable.

SOLID Principles

- Acronym of acronyms:
  - SRP: Single Responsibility Principle
  - OCP: Open-Closed Principle
  - LSP: Liskov Substitution Principle
  - ISP: Interface Segregation Principle
  - DIP: Dependency Inversion Principle
- Basically a set of principles for object-oriented design (with focus on designing the classes).
First Pass at Understanding SOLID

- SRP: “A class should have one, and only one, reason to change”.
- OCP: “You should be able to extend a class’s behavior, without modifying it”
- LSP: “Derived classes must be substitutable for their base classes.”
- ISP: “Make fine grained interfaces that are client specific.”
- DIP: “Depend on abstractions, not on concretions.”

SRP: Cont’d

- Solution: Take the purely computational part of the Rectangle class and create a new class “Geometric Rectangle”.
- All changes regarding graphical display can then be localized into the Rectangle class.

SRP: Another example

- Modem: dial(), hangup(), send(), recv(), ...
- However, there are two separate kinds of functions that can change for different reasons:
  - Connection-related
  - Data communication-related
- These two should be separated.
- Recall that “Responsibility” == “a reason to change”.
**SRP: Summary**

- “SRP is the simplest of the principles, and one of the hardest to get right.”
- We tend to join responsibilities together.
- SRP says we need to go against this tendency.

**OCP: Open-Closed Principle**

- “All systems change during their life cycles.” (Ivar Jacobson).
- “Software entities should be open for extension, but closed for modification.” (variation on Bertrand Meyer’s idea).
- Goal: avoid a “cascade of changes to dependent modules”.
- When requirements change, you extend the behavior, not changing old code.

**OCP: Abstraction is Key**

- Bad design: need to change client code when new kinds of server needed.
  ![Client Server Diagram](#)
  
  - Good design: can extend to new types of servers without modifying client code.
    ![Client Abstract Server Diagram](#)

**OCP: Data-Driven Approach**

- In many cases, complete closure (closure to modification) may not possible.
- Data-driven approach can be taken to minimize and localize changes to a small region of code that only contain data, not code.
- For example, there can be a table that contains a specific ordering based on the requirements, where the requirements are expected to change.
OCP: Foundation for Many Heuristics

OCP leads to many heuristics and conventions.

- Make all member variables private.
- No global variables, EVER.
- Run time type identification (e.g., dynamic cast) is dangerous.
- etc.

LSP: Liskov Substitution Principle

- “Functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it.” (original idea due to Barbara Liskov).
- Violation means the user class’s need to know ALL implementation details of the derived classes of the base class.
- Violation of LSP leads to the violation of OCP.

OCP: Summary

- OCP is “at the heart of OOD”.
- Simply using an OOP is not enough: Need dedication to apply abstraction.
- OCP can greatly enhance reusability and maintainability.

LSP: Example

Rectangle Class ← Square Class

- Problem: setWidth(), setHeight() in Rectangle class assumes w and h are independently settable.
- When Square class is used where Rectangle class is called for, behavior can be unpredictable, depending on implementation.
- Want either setWidth() or setHeight() to set both width and height in the Square class.
- LSP is violated when adding a derived class requires modifications of the base class.
LSP: Lessons Learned

- Cannot assess validity of a class by just looking inside a class: We must see how it is used.
- “ISA relationship pertains to behavior”, extrinsic, public behavior!
  - Square is a Rectangle, but they behave differently, seen from the outside.
- For LSP to hold, ALL derived classes should conform to the behavior that the clients expect of the base classes.

ISP: Interface Segregation Principle

- “Clients should not be forced to depend upon interfaces that they do not use.”
- Avoid “fat interfaces”.
- Fat interfaces: interfaces of a class that can be broken down into groups that serve different set of clients.
- Clients depending on a subset of interfaces need to change when other clients using a different subset changes.

LSP: Summary

- LSP is an important property that holds for all programs that conform to the Open-Closed principle.
- LSP encourages reuse of base types, and allows modifications in the derived class without damaging other components.

ISP: Example

- Bad design
- Good design

Clients that use Door or TimerClient access only those specified interfaces.
ISP: Summary

• Should avoid interfaces that are not specific to a single client.
• Fat interfaces cause inadvertant coupling between unrelated clients.

DIP: Dependency Inversion Principle

• “A. High level modules should not depend upon low level modules. Both should depend upon abstractions.”
• “B. Abstractions should not depend upon details. Details should depend upon abstractions.”
• DIP is an out-growth of OCP and LSP.
• “Inversion”, because standard structured programming approaches make the higher level depend on lower level.

DIP: The Problem

• Bad design:
  – Hard to change (rigidity)
  – Unexpected parts break when changing code (fragility)
  – Hard to reuse (immobility)
• Cause of bad design:
  – Interdependence of the modules
  – Things can break in areas with NO conceptual relationship to the changed part.
  – Dependent on unnecessary detail.

DIP: Example

Copy(): uses ReadKeyboard() and WritePrinter(char c);

• Copy() is a general (high-level) functionality we want to reuse.
• The above design is tied to the specific set of hardware, so it cannot be reused to copy over diverse hardware components.
• Also, it needs to take care of all sorts of error conditions in the keyborad and printer component (lots of unnecessary details creep in).
DIP: Diagnosis of Copy()

- Module containing high level policy (Copy) is dependent upon low level detailed modules it controls (WritePrinter, ReadKeyboard).

- Good design:

  Encourages reuse of higher level policies.

DIP: Layering and Better Layering

- Bad Design

  Policy Layer
  Mechanism Layer
  Utility Layer

- Good Design

  Policy Layer
  Abstract Mechanism Interface
  Abstract Utility Interface
  Utility Layer

Policy layer not dependent on lower levels, thus can be reused.

DIP: Another Example

- Bad Design

  Button
  Lamp

  When button changes, lamp has to be at least recompiled. Cannot reuse button for different device.

- Good Design

  Abstract Button
  Abstract ButtonClient

  Can further introduce LampAdapter.

DIP: Summary

- DIP promises many benefits of OO paradigm.

- Reusability is greatly enhanced by DIP.

- Code can be made resilient to change by using DIP.

- As a result, code is easier to maintain.
SOLID Principles: Summary

- Help manage dependency.
- Improved maintainability, flexibility, robustness, and reusability.
- Abstraction is important